Requested Patent: WO9710543A1

Title: AUTOPILOT DYNAMIC PERFORMANCE OPTIMIZATION SYSTEM ;

Abstracted Patent: WO9710543 ;

Publication Date: 1997-03-20 ;

Inventor(s): SHEETS KITRICK ;

Applicant(s): MCSB TECHNOLOGY CORP (US) ;

Application Number: WO1996US14540 19960911 ;

Priority Number(s): US19950003561P 19950911 ;

IPC Classification: G06F9/44 ;

Equivalents: AU6973196, AU7360296, WO9710548 ;

ABSTRACT:

The AutoPilot (54) performance optimization module is a part of the Performance Assistant family (52) which is designed to dynamically optimize and balance the performance of multiprocessor computer systems. AutoPilot (54) utilizes proactive hardware monitoring capabilities supplied through the Performance Assistant architecture to monitor a computer system's workload and make performance adjustments in real time.

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/44	A1	(11) International Publication Number: **WO 97/10543** (43) International Publication Date: 20 March 1997 (20.03.97)

(21) International Application Number: PCT/US96/14540

(22) International Filing Date: 11 September 1996 (11.09.96)

(30) Priority Data:
60/003,561 11 September 1995 (11.09.95) US

(71) Applicant: MCSB TECHNOLOGY CORPORATION [US/US]; 4330 Golf Terrace, Eau Claire, WI 54701 (US).

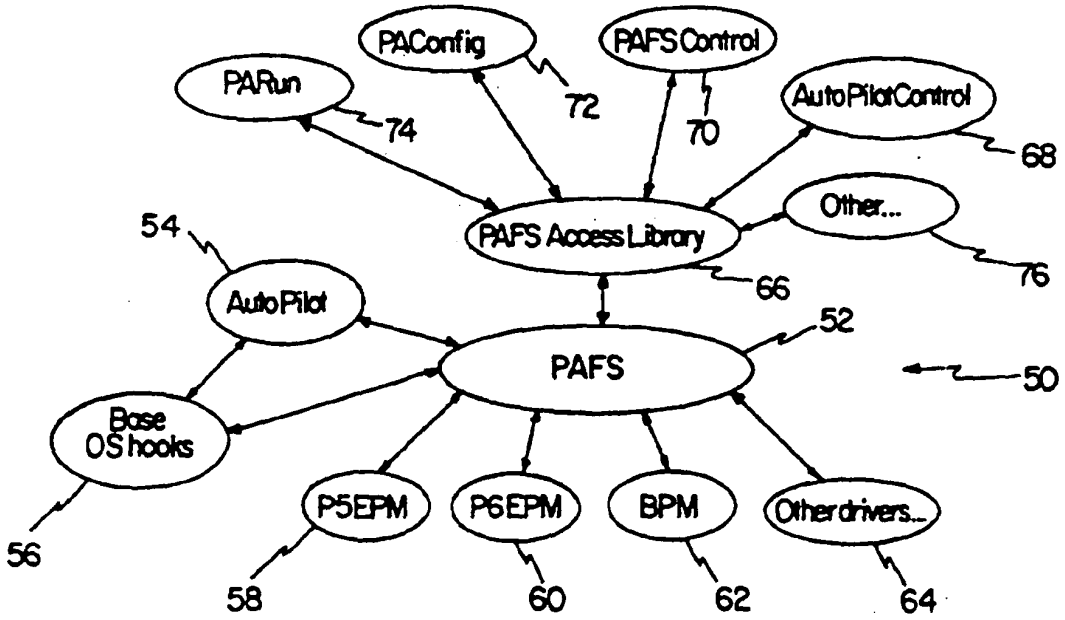
(72) Inventor: SHEETS, Kitrick; 2179 Aebly Road, Chippewa Fall, WI 54729 (US).

(74) Agents: STANGA, Paul, W. et al.; Patterson & Keough, P.A., 1200 Rand Tower, 527 Marquette Avenue South, Minneapolis, MN 55402 (US).

(81) Designated States: AU, CN, JP, KR, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: AUTOPILOT™ DYNAMIC PERFORMANCE OPTIMIZATION SYSTEM



(57) Abstract

The AutoPilot (54) performance optimization module is a part of the Performance Assistant family (52) which is designed to dynamically optimize and balance the performance of multiprocessor computer systems. AutoPilot (54) utilizes proactive hardware monitoring capabilities supplied through the Performance Assistant architecture to monitor a computer system's workload and make performance adjustments in real time.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AM	Armenia	GB	United Kingdom	MW	Malawi
AT	Austria	GE	Georgia	MX	Mexico
AU	Australia	GN	Guinea	NE	Niger
BB	Barbados	GR	Greece	NL	Netherlands
BE	Belgium	HU	Hungary	NO	Norway
BF	Burkina Faso	IE	Ireland	NZ	New Zealand
BG	Bulgaria	IT	Italy	PL	Poland
BJ	Benin	JP	Japan	PT	Portugal
BR	Brazil	KE	Kenya	RO	Romania
BY	Belarus	KG	Kyrgystan	RU	Russian Federation
CA	Canada	KP	Democratic People's Republic of Korea	SD	Sudan
CF	Central African Republic			SE	Sweden
CG	Congo	KR	Republic of Korea	SG	Singapore
CH	Switzerland	KZ	Kazakhstan	SI	Slovenia
CI	Côte d'Ivoire	LI	Liechtenstein	SK	Slovakia
CM	Cameroon	LK	Sri Lanka	SN	Senegal
CN	China	LR	Liberia	SZ	Swaziland
CS	Czechoslovakia	LT	Lithuania	TD	Chad
CZ	Czech Republic	LU	Luxembourg	TG	Togo
DE	Germany	LV	Latvia	TJ	Tajikistan
DK	Denmark	MC	Monaco	TT	Trinidad and Tobago
EE	Estonia	MD	Republic of Moldova	UA	Ukraine
ES	Spain	MG	Madagascar	UG	Uganda
FI	Finland	ML	Mali	US	United States of America
FR	France	MN	Mongolia	UZ	Uzbekistan
GA	Gabon	MR	Mauritania	VN	Viet Nam

AUTOPILOT™ DYNAMIC PERFORMANCE OPTIMIZATION SYSTEM

RELATED APPLICATIONS

This application is a formal application of a part of the Provisional Application filed on September 11, 1995 and assigned Serial Number 60/003,561.

FIELD OF THE INVENTION

The invention relates generally to capacity and workload optimization of data servers and workstations. More specifically the present invention pertains to performance enhancement, tuning and optimization software to enable the best possible use of an underlying system hardware.

DESCRIPTION OF RELATED ART

Data servers and workstations are well known in the art. The computing industry has moved toward distributive computing networks comprised of heterogeneous workstations, connected together in an open system network. As a result, there has been an ever increasing demand for more powerful and faster data servers to service both the distributive data management and processing requirements of users within the network. In response to this ever increasing demand, multiprocessing data servers, such as described in U.S. Patent No.'s 5,355,453, and 5,163,131 to Row, et al., have been developed.

Unlike traditional mainframe computer systems and subsystems for mainframe computers which operate in a captive, homogenous environment, network data servers must be capable of responding to resource requests from a wide variety of users in a heterogeneous network environment. The data server must respond to these requests in an efficient, yet distributed, manner without any type of central control from a mainframe computer. As such, the problems and demands imposed on

the design of a system architecture for a network data server are significantly different than for a traditional mainframe computer system and subsystem.

While the design of existing data servers has been sufficient to accommodate the demands of most users in a network computer environment, it would be advantageous to provide a performance assistant architecture for a data server to enable tuning and optimization of the performance, scalability, robustness and maintainability required for the ever-increasing demands of network client-server applications.

BACKGROUND OF THE INVENTION

Performance optimization and tuning require specialized tools and highly skilled personnel. Specifically, long hours are spent collecting and analyzing the output of these tools to uncover hidden inefficiencies and bottlenecks affecting system hardware. This approach is labor-intensive and quickly degrades into a diminishing rate of return.

Over the last several years, many advances have been made in the design of high performance computer systems. An ever increasing need for high performance has resulted in the development of new approaches in system and machine design. Some of these new approaches include schemes which join multiple processors in a single system which cooperate in the execution of the user's application workload. While there are several techniques which can be used to join multiple processors within a system, the most common technique used today is that of Symmetric Multiprocessing (SMP). In fact, SMP is being offered by many computer manufacturers as entry level systems. Thus, the multiprocessor environment is becoming evermore complex. In light of this development, diagnosis and evaluation of multiprocessor operations should be compatible and keep pace with the current trend in the art.

The present state of the art is such that evaluation and diagnosis to optimize performance is rather complex and requires intensive labor and various tools. In order to get maximum performance from a software

implemented in a computer or an application executing on any system, it is important to understand how the software code interacts with the underlying hardware components. A successful performance evaluation should, at a minimum, be able to examine how each of the hardware components interact with the software implemented therein.

Techniques for optimizing performance are now well known, but as indicated hereinabove, the required intensive labor and the associated tools make it impractical to routinely and efficiently use these techniques in an increasingly complex environment of modern servers and workstations.

To simplify and significantly enhance the process of system performance, evaluation and tuning there is a need to integrate software in a performance assistant (PA) architecture. Further, it is required that such software must provide a set of compatible tools. The software and associated tools must further enable a generic interface for the collection of performance data and the correlation of this data to executing tasks within a system hardware.

Accordingly, any optimization system and software dedicated to diagnose and tune data servers and workstations, must employ efficient and reliable tools. Thus, there is a need for an interactive and diagnostic system, which is compatible with the evolving complex environment of servers and workstations, to enable a real-time, comprehensive and automatic optimization of task/workload allocation for data servers and workstations.

SUMMARY OF THE INVENTION

The AutoPilot™ dynamic performance optimization system of the present invention utilizes information obtained from passive hardware monitoring systems to dynamically optimize the allocation of tasks or resources by the operating system. It is a software-based tool which provides a reliable means for system performance monitoring and tuning. More specifically, the present invention enables evaluation and tuning of

system hardware workloads and task allocations, in minute details, in their existing environment without undue interruptions or down time.

The concept of AutoPilot™ is based on a "missing link" in an operating system's ability to effectively manage the workload which is presented to it. Prior to AutoPilot™, operating systems have scheduled tasks based solely on their priority and without regard to the impact that they have on the underlying hardware and, more importantly, other tasks within the system.

To date, operating systems have relied heavily on advanced compiler techniques and advances in system architectures to improve the performance of an application workload. Now that microprocessors and other system components support the ability to monitor application performance in real-time, a whole new door has opened in the areas of system performance optimization. Having the ability to monitor application behavior allows the operating system to provide a new dimension to the process of effectively scheduling tasks within the system.

The performance penalties associated with an inefficient execution of tasks on multiprocessor systems are well known in the art. Techniques such as affinity scheduling, which attempts to schedule tasks on processors on which they may still have valid cache contents, is fairly common in most operating systems today. However, while the concept of affinity scheduling is valid, the implementation is normally based on the use of arbitrary time limits between task executions to determine whether cache contents may still be valid. No real measurement is done to determine whether or not this is the case.

AutoPilot™ is designed to fill this void within the operating system. It uses the real-time performance data collected on its behalf by the Performance Assistant (PA) architecture to provide dynamic optimization of a system's application workload. With this information, AutoPilot™ can make informed decisions about the impact of applications on the underlying hardware and other concurrently executing tasks. In this way, AutoPilot™ can ensure the optimal use of available hardware resources

and allow the system to operate at it's full potential.

Accordingly, AutoPilot™ is a system level tuning feature. It is implemented in a PA architecture. The PA architecture provides a strong base for tuning applications and includes an extensible framework which can support the development of novel system level tuning features. The first of these is the AutoPilot™ System Performance Optimizer. AutoPilot™ uses real-time performance data collected through performance assistant file system (PAFS) to ensure that the current application workload makes an optimal and best possible use of the system hardware. Autopilot™ collects dynamic system performance data non-intrusively and in real time. This data can then be used to tune system software, hardware or applications, both automatically and manually to get the best possible performance from the system.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a prior art block diagram of a symmetrical multiprocessing (SMP) computer system.

Figure 2 is a functional view of a superscalar processor.

Figure 3 is a block diagram of a pipelined processor architecture.

Figure 4 is a structure depicting the performance assistant environment.

Figure 5 shows a structure of a task dispatch queue of a typical operating system.

Figure 6 is a block diagram depicting the functional aspects of a bus performance monitor (BPM) .

Figure 7 is a sample work multiprocessor workload.

Figure 8 is a diagram depicting an operating system process selection.

Figures 9A and 9B depict the implementation of AutoPilot™ for task selection among a plurality of processors.

Figure 10A is a graphical representation of the effect of resource conflict on processing efficiency.

Figure 10B is a graphical representation of scalability in a simple SMP system.

Figures 10C and 10D show processing efficiency with I/O load.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is advantageously applicable to data servers and workstations. In order to provide a basis for understanding the present invention and its advantageous features, prior art computer systems and relevant processor architecture are discussed hereinbelow.

Figure 1 is a standard symmetric multiprocessor (SMP) 10 which is typical of most server systems available today. Generally such systems consist of a number of processors which reside on a shared system bus. In Figure 1, CPU 12 depicts a plurality of such processors. CPU 12 includes a first level (L1) 14 cache system. Further, each processor has a second level (L2) 16 cache memory and communicates with other processing elements through a single shared global memory. Memory 18, Network 20 and disk I/O 22 share an interconnect path 23 with CPU 12. Similarly data path 23' interconnects CPU 12, memory 18, network 20 and disk I/O 22. Protocol and consistency is maintained by special algorithms designed to monitor the system bus and maintain L2 cache 16.

Much work has been done over the last several years to enhance the performance of microprocessor architectures. This has led to various architectural advances in microprocessor design. Techniques such as parallel execution, pipelining and speculative processing are becoming common in most microprocessors built today.

Parallel execution utilizes multiple functional units into a chip and installs multiple operations on a single processor cycle. A processor with such capability is called "super scalar". Figure 2 is a simplified block diagram of a superscalar processor architecture 30. The functional aspects include instruction stream 32, functional units 34 and system interface 36.

A typical prior art example of pipelining is shown in Figure 3. Pipelining techniques are implemented to make up for performance losses

which occur in a processor when instructions take multiple cycles to execute. In a pipelined architecture instructions move through the processor one stage at a time until the entire instruction has been executed. Figure 3 provides a functional view of such an architecture. Instruction stream 40 passes through pipeline stages 42 to system 44 and the cycle is repeated as apparent. This technique does not reduce the amount of time required to execute a given instruction. However, it permits execution time of the instruction to be overlapped thereby reducing overall circuitry occupancy for a given period time.

Although pipelining enhances the performance of some codes, there are limitations to this technique. A major problem occurs when the processor branches to a new piece of code and experience pipeline stall. Techniques such as branch prediction and speculative execution have been developed to examine the execution path of an application and predict what code segment will be needed in the immediate future execution.

Currently, commonly available microprocessors provide performance levels surpassing that which was available only in sophisticated super computers a few years ago. This advancement in processor speeds makes it inevitable that any discussion of a machine's performance would begin with a review of the capabilities of its CPU.

While clock frequencies on the latest CPU's continue to increase at a tremendous rate, this is not the only trick processor designers are using to increase the performance of their chips. Tricks such as parallel execution, pipelining and speculative processing are becoming common in most microprocessors built today.

Applications which can fit within the on-chip cache can normally execute at nearly the full clock rate of the processor. However, since these caches are generally of relatively modest size (8Kbytes - 16 Kbytes), few codes fit well into this space. For this reason, most of today's systems include second level (L2) 16 cache memories (Refer to Figure 1).

L2 caches 16 are usually several times larger than their on-chip counterparts (256Kbytes - 2Mbytes). This increased size makes it possible to

dramatically increase the amount of data and instructions kept in close proximity to CPU 12. In addition, the amount of bus traffic used by CPU 12, for example, to access main memory is reduced making more cycles available to other system components.

To reduce the burden on memory 18 even more, L2 caches 16 incorporate a write-back data write policy. This policy allows the actually transfer of modified data to main memory to be delayed until the cache line is reused or explicitly flushed. This allows the processor to manipulate cached data locally without impact on the rest of the system.

While L2 caches 16 have gone a long way to reduce the performance penalties associated with off processor accesses, they are still only one component of the memory hierarchy that affects system performance. Regardless of the application, eventually the processor needs to access data beyond the cache boundaries in order to perform useful work. For this reason, the design of a system's subsystem is a key component in the machine's performance. The system's memory must be able to supply data at a sufficient rate or the performance of the system as a whole will suffer.

To illustrate this point, consider the system in Table I with a memory hierarchy in which 50% of the memory accesses are satisfied by the level 1 cache and another 40% are satisfied by the second level cache. The balance of accesses to memory (10%) must be satisfied from the system's main memory. Let's also assume that a hit in the first level cache costs 1 CPU cycle on average, an access to the second level cache costs 3 CPU cycles, and an access to the main memory costs 15 system bus cycles. As is commonly the case, it is assumed that the system bus runs at some fraction of the processor clock rate. In this example, let the bus be clocked at 1/2 the rate of the system processors. This means that the 15 system bus cycles for a memory access actually translates into 30 CPU cycles.

From the above example, we can calculate the average cost of a memory access for the machine:

Memory level	Percent	Cycles	Total
L1 cache	50%	1	0.5
L2 cache	40%	3	1.2
Main memory	10%	30	3
Total			**4.7**

Table I. Memory access times

This example illustrates how a relatively small number of access to main memory can have a major impact on the average memory access time. Even though the cache subsystem achieves a 90% hit rate, the high latency of the memory subsystem causes the average latency of all memory accesses to increase from less than two to almost five CPU cycles. This example shows that while an effective caching hierarchy is very important to a high performance system's design, the impact of the main memory subsystem's performance still has a large impact on the overall performance of the system.

While it is important to design a CPU and memory subsystem which can support a high computational capacity, a balanced system design must have an I/O subsystem capable of supporting this compute load. Ignoring the importance of the I/O subsystem in such a machine certainly limits its ability to support a diverse application load. In this section some of the important issues in the design of a production class I/O subsystem are addressed.

In environments where many client systems are attached to a database server, each issuing simple transactions (e.g. bank tellers), it is important to maintain a low response time for each of the user's requests. This translates directly into an I/O subsystem which can service each of these requests with a very low latency.

Likewise, in environments where client machines are each

requesting large amounts of data from the server system (e.g. for visualization, video on demand, or decision support), users demand a high sustained data rate. This translates directly to a requirement on the server to be capable of sustaining a high network and disk throughput rate as well as low access latencies.

A key component in the design of a high performance I/O subsystem is the existence of a system and I/O bus interconnect which can sustain high transfer rates to and from main memory. If the interconnect allows high transfer rates on a lightly loaded system but degrades rapidly as system load increases, the system as a whole will not be suitable for use in large database of file server applications. For this reason, it is important to design an I/O subsystem that balances and complements the rest of the system. As I/O demands increase, the I/O subsystem must be capable of sustaining these requests while not overloading the system bus with unnecessary overhead.

As an example of how these components interact with one another, consider again the system in Figure 1. This architecture is very typical of most server systems available today. As can be seen from this figure, I/O requests move from the peripheral devices to main memory at which point the CPU can examine and manipulate the data. After completing its work, the CPU schedules to have the data written back to disk or out to the network.

This scenario works effectively while system load is moderate, however, as load increases, conflicts between the various system components competing for access to memory increase rapidly reducing the available bus bandwidth.

Traditionally, the approach used to solve this problem has been to increase the capacity of the system bus and main memory. Unfortunately, this is only possible up to a point. Increasing the bus and memory bandwidth is very expensive and usually involves the use of cutting edge technologies. All of this expense doesn't necessarily have a linear pay back either. A performance increase of 50% for both the memory and system

bus would likely double their cost.

In the context of the present invention software plays an important role in the optimization of data servers and workstations. On any server it is important that applications run as efficiently as possible. Generally, efficiency is dependent on utilization of processor resources and cache. As indicated in supra, modern processors are equipped with multiple functional units. These functional units and their associated pipelines can be utilized to allow the processor to deploy all of its available resources on the application. Further, the interaction of the application with the processor's cache architecture is also crucial to the performance and overall system efficiency.

Furthermore, interprocessor communication (IPC) is becoming a common place occurrence in servers and network systems. This becomes even more common as applications are designed to take advantage of lightweight thread interfaces, available in most modern operating systems, to break application tasks into smaller pieces to communicate or consolidate data. When such code is executed in multiprocessor systems the need to avoid cache thrashing conditions becomes critical.

Moreover, system performance is dependent upon operating system software. The operating system controls the utilization of hardware resources and needs to be efficient in managing this allocation function. In multiprocessor operating system, for example, internal data structures are protected from simultaneous update through the use of a variety locking algorithms. At the base of these locking routines are hardware primitives which support atomic memory update transactions. To protect structures which require frequent and fast updates locking, routines are used which loop on the lock variable until it is acquired exclusively (spin locks). Sleep locks or semaphores are used when a lock is not available. Spin locks are used for fine grained locking and semaphores are used for coarse grained locking.

Yet another performance optimization which is often performed on operating systems is hand coding or algorithmic optimization of the most

commonly executed code sections.

The present invention includes tuning methods that substantially advance the state of the art. Traditional system performance tuning methods include both software based and hardware based tuning tools. Generally, the software based tuning tools allow developers to trace the execution path of their code and obtain code path statistics on the performance characteristics of an application or system. This information is used to determine which components in the system or application are consuming the most execution time. When such information is determined, the tools can be used to optimize the components. Similarly, Windows/NT also provides several performance tuning tools. Further, compilation systems which are capable of performing feedback directed recompilation are available.

To examine system performance related activity at the hardware level, in circuit emulation (ICE) devices or logic analyzers are sometimes used. These devices capture signals on various buses within the system and store and/or display the results of that trace. Any information which appears on a processor or system bus can be examined with these devices. Although these systems are useful for monitoring the low level details of the system hardware, the devices are generally expensive.

While each of the tools in the prior art discussed in supra are useful for the specific tasks for which they are designed, there are circumstances where some of these tools become impractical or impossible to use. For example, current tools are less than optimal for evaluating and tuning the performance of applications and the system as a whole. This is primarily due to intrusiveness, coarse level of coverage, disjoint tool set and limited end-user use. Probably the most limiting factor on current hardware and software performance tuning techniques is the limitation placed on the end-user. Most tuning devices are designed for use in development or laboratory environments.

Referring now to Figure 4 a structure of the Performance Assistant (PA) architecture 50, in which the present invention is implemented is

shown. Specifically, performance assistant file system (PAFS) 52 is shown as the central connection point of PA architecture 50. PAFS 52 acts as a clearing house for all performance data collected from the hardware while supplying the critical correlation between that data and applications running on the system. PAFS 52 is connected to Autopilot 54. Base operating system hooks 56 includes a two-way communication with Autopilot 54 and PAFS 52. Further, Engine Performance Monitors (EPM) 58 and EPM 62 are in a two-way data communication with PAFS 52. Bus Performance Monitor (BPM) 62 provides processor performance feedback and is also in a two-way communication with PAFS 52. PAFS 52 enables additional drivers 64 to be added depending on the complexity of the structure and the components involved therein. PAFS 52 is also in a two-way communication with PAFS access library 66. PAFS access library 66 enables and provides a two-way communication with AutoPilot control 68, PAFS control 70, PA configuration 72 and PARun 74 and other controls 76.

Figure 5 shows a typical operating system thread dispatch. Normally, the operating system (e.g. Unix or Windows/NT) will keep runable tasks on some number of dispatch queues 76 depending upon their execution priority. Priority determination is based on a number of factors including resources required by task 78, elapsed time since the task last executed, etc. Tasks with equal priorities are placed on the same list. When one or more processors 80 become available, the operating system chooses a task from the highest priority task list and starts executing it on that processor.

As discussed earlier, tasks 78 executing on one processor in a multiprocessor system have a dramatic impact on other tasks 78 which may be executing concurrently on other processors 80. If tasks 78, which require continuous access to some hardware resource (e.g. system memory 82), are executed concurrently both will suffer. In addition, the extra bus and memory 82 bandwidth consumed by these tasks will also leave less bandwidth available for background tasks such as disk I/O. This can have

a direct impact on the I/O throughput provided by the system.

If the operating system was capable of determining what hardware resources are used by each thread within the system, situations where certain hardware components are overloaded could be avoided and system throughput could be increased. Microprocessor and system manufacturers are beginning to provide components which are capable of performing real-time monitoring of hardware performance. With this hardware in place, it is possible to build sophisticated software interfaces which can take full advantage of these facilities. This is the basis for the development of Performance Assistant Family of products. Using the Performance Assistant as a base, it is possible to dynamically balance the hardware requirements of the application workload and ensure optimal use of the system's hardware resources.

As discussed hereinabove, the PA 50 architecture provides a strong base for tuning applications. Referring now to Figure 4, AutoPilot 54 and AutoPilot control 68 are some of the prominent features of the PA 50 architecture which enable performance optimization. AutoPilot 54 uses real-time performance data collected through PAFS 52 to ensure that the current application workload makes the best possible use of the system hardware. By monitoring the execution patterns of applications running on a server, AutoPilot 54 can predict over-commitment of critical hardware resources and manipulate the workload so as to avoid such bottlenecks.

Figure 7 is a sample multiprocessor system workload showing exemplary processors P1, P2, P3 and P4 interconnected to each other and also connected to memory 100. The processors are engaged in outstanding workload made up of tasks with varying system bus requirements. Each processor is running a task which it has acquired from a shared queue of outstanding jobs. The numbers to the right of and above each waiting task correspond to the percentage of the system bus that the task requires during its execution. At the right of the Figure, the aggregate system bus load created by the active tasks in the system is shown.

As discussed hereinbelow, AutoPilot™ uses a new approach in system performance tuning to ensure that the user obtains the maximum performance possible from the available system hardware. Techniques developed by Chen allow a much higher utilization of available system resources which translates directly into performance dividends for the end user.

The most obvious impact that additional processors have on the system is the additional load that they place on the system bus and memory. As discussed hereinabove, today's servers employ caching techniques which are designed specifically to reduce a processor's reliance on main memory accesses. Table 1 above shows the impact that a relatively small percentage of main memory accesses can have on the performance of an application.

Number of CPUs			1	2	3	4	5	6	7	8	9	10
Memory Level	Percent of accesses	Cycles per access	Total	10%	20%	30%	40%	50%	60%	70%	80%	90%
L1 and below	50%	1	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
L2 cache	40%	3	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2	1.2
Main Memory	10%	30	3.0	3.3	4.2	5.7	7.8	10.5	13.8	17.7	22.2	27.3
TOTAL CPI			4.7	5.0	5.9	7.4	9.5	12.2	15.5	19.4	23.9	29.0

Table 2.

An exemplary SMP design and the impact of additional processors on memory access is depicted in Table 2. The lower left portion of the table (light shading) corresponds to the information in Table 1. The 'Number of CPUs' row at the top of the table represents the number of processors

within the system. In this example, we maintain the access distribution presented in Table 1 (I.e. 50% of references occur on-chip, 40% are satisfied by L2, and the remaining 10% must be satisfied by main memory).

As can be seen in this table, the access times to L1 and L2 cache remain constant as the number of processors is increased since they are local to each processor and act (mostly) independently of other processors within the system. However, as the number of processors in the system increases, so does the access time to main memory. This is because there is a single, access path to memory which needs to be shared among each of the compute nodes.

In this example it is assumed that 10% of the accesses made by a processor need to acquire the system bus and access main memory. In a single processor case, the processor can be assured that there will be no competition for this access. However, as processors are added, this is no longer the case. In the table, we assume that each additional processor follows the same model as the single processor case and requires access to the bus 10% of the time. In the two processor case, it is assumed that since there is another processor with similar execution characteristics sharing the bus, we assume that there will be a conflict for bus on 1 out of 10 accesses. This pattern continues as processors are added to the system. The row of values directly beneath the number of CPUs in the new portion of the table correspond to the percentage of bus accesses which conflict as the degree of parallelism within the system increases.

As the table shows, even though the increase in parallelism within the system has no effect on the processor's ability to satisfy 90% of an application's accesses locally, the increased latency to main memory induced by contention for system bus access has a dramatic impact on the application's performance as measure by Cycles Per Instruction (CPI).

It is easier to see this effect in the graph in Figure 10A. This graph shows very clearly what happens to computer efficiency as processors are added to the system. In this graph, two lines are provided which compare the "Ideal" execution characteristics (e.g. no memory resource contention)

of the application in this environment with the effect that increased competition for memory access has on the application. As can be seen in this graph, the processing efficiency of this application decreases by 4 times when executed in an 8 CPU SMP system environment. This effect is even more interesting when we consider that contention is for only 10% of the processor accesses. A less effective cache architecture would yield an even more dramatic change in processing efficiency.

Figure 10A, shows the performance of the SMP system from the perspective of application processing efficiency. Another way of looking at SMP compute effectiveness is to consider the system's "scalability". Scalability considers how effective the addition of processors to a system is at handling an application workload. If a system is capable of completing a task with two processors in 1/2 the time required by the same system with a single processor, the system is said to scale linearly. Linear scalability is the goal of all multiprocessors.

Figure 10B shows the scalability of the system in the example. As shown in this example, the system scales fairly well to three processors. However, as the fourth, fifth and sixth processors are added, the effective processing increase of each additional processor begins to flatten out dramatically. When the seventh and eighth processors are added to the system, the performance benefit is negligible.

To this point the focus has been primarily on the impact that the system's processors have on contention for memory access. However, it is not realistic to consider the performance of an SMP system without examining the requirements of the I/O subsystem and the impact that it has on overall system performance.

Referring again to Figure 1, we see that the processors in this SMP system share a single communication path 23 to main memory with the I/O devices (network and disk 22). Although this diagram shows a single link to the disk I/O subsystem, it is common for systems to have multiple parallel I/O channels (or bridges) which support connection to high performance I/O devices.

To demonstrate the effect of an I/O load on the performance and scalability of the system form the previous example, let's assume that the running application creates a background I/O load which consumes 3% of the system bus bandwidth.

As can be seen in Figures 10C and 10D, the additional traffic caused by the addition of the background I/O load has a noticeable impact on overall system performance. From the first graph in this figure, we can see that the processing efficiency of the application decreased by about 20% with the addition of I/O. Likewise, the scalability of the system as shown in the second graph was noticeably reduced. This example shows the impact of a fairly low background I/O load on the system. A moderate or high level of background traffic will have a much more dramatic effect on overall performance.

Accordingly, Figure 7 shows the current system commitment is at 90% based on the currently executing task requirements. This shows that the bus system is near saturation assuming maximum utilization to be at 100%.

As is the case in a timesharing operating system like Unix or Windows/NT, after a certain amount of run time, tasks are taken off of a processor and other tasks are allowed to run. This ensures that available processor resources are allocated and distributed across the available work list. This is generally accomplished, on most operating systems, by taking the first available task from a queue which contains tasks with the highest run priority in the system. Figure 8 shows an operating system process selection depicting the proceeding process of allocation. As can be seen in Figure 8, randomly selecting the next available task causes an over commitment of system bus resources or bus saturation. Therefore, the tasks running on each of the processors are now in a mode where they are committing a significant portion of their time competing with other CPUs for system bus access. This generally results in the CPUs being stalled and eventually leads to overall system performance degradation.

Figures 9A and 9B show task selection with AutoPilot 54. The

Figures depict a task selection method which is intelligent and avoids the problems outlined hereinabove during task selection and allocation among CPUs. The Figures show how a task selection can be efficiently and effectively implemented. Primarily, the next thread to be dispatched onto an available processor is based upon consideration of the thread's impact on the performance of the system as a whole. Further, from Figures 9A and 9B, it is clear that by considering task performance characteristics, saturation of the bus can be avoided and a better utilization of the system hardware can be achieved. Thus, without changing the system hardware or architecture, significant improvements in overall system performance can be realized. The present invention advantageously implements Autopilot 54 to provide performance enhancements.

Furthermore, AutoPilot control 68 tool is used for system specific tuning of the AutoPilot optimization system. The implementation and behavior of AutoPilot is very much dependent upon the performance characteristics of the server on which it is running. In addition, each site has certain rules about what type of processing is most important for the operation. For example it may be the case that on one system the user wants to be guaranteed that there is always bandwidth available to handle I/O bound tasks as they appear. In this case AutoPilot would be configured to reserve some bandwidth for this purpose. On other systems, computer bound tasks may be most important. In this case, most of the bandwidth is allowed to be taken by processing elements. Thus, the AutoPilot control tools enable the user to have flexibility in making decisions relating to specific installations. Once configured, AutoPilot works to keep the system within the specified processing parameters and bounds.

Thus, AutoPilot 54 is a software module which plugs directly into the host operating system to ensure that the underlying hardware is used in the most optimal manner. As discussed hereinabove, AutoPilot 54 monitors the activity of each task within the system and makes task allocation and scheduling decisions based on this information. AutoPilot 54 can be thought of as a layer in the server toolset architecture as is

represented in Figure 4. At one level exists hardware components specifically designed for collecting performance data. These components include PAFS 52, EPM 58 and 60, BPM 60 and other drives 64. The Engine Performance Monitors (EPMs) include a set of specialized registers within the Pentium processor which collects statistics on the execution behavior of tasks running on the CPU at any time. As discussed hereinabove, information available from these registers includes cache hit statistics, instruction pipeline utilization, functional unit utilization and the like. Base OS hooks 56 are designed to give AutoPilot 54 access to critical task selection points within the operating system.

AutoPilot 54 is controlled via AutoPilot control 68. The controls are tunable variables configurable through the standard UnixWare idtune interface. The interface controls the point at which the system bus is considered saturated. If the current bus utilization is below the saturation level AutoPilot 54 will act similar to a default system. In other words each thread will be run in turn without regard to its system bus requirements. If the system bus load exceeds the tunable values, AutoPilot 54 will attempt to find the best available thread to run based on the bus load of the threads at the best system task priority, depending upon the setting of the variable. In the alternate, depending upon the setting, AutoPilot 54 may opt to idle a processor if no thread can be found which will fit below the specified system bus threshold.

Yet another tunable interface of AutoPilot controls the rate at which the bus load for a thread is adjusted over its execution lifetime.

Thus, AutoPilot 54 provides novel optimization features by implementing sophisticated performance evaluation and tuning tools which are user-friendly. It is generally installed in a PA 50 and enhances the performance and coordination of the underlying hardware resources. In the interest of simplicity the present invention and the concepts disclosed herein are shown implemented in an SMP environment. However, these are exemplary embodiments and applications of the invention and are construed to be non-limiting. The AutoPilot dynamic

performance optimization system of the present invention is generally applicable and could be used in other multiprocessors and architectures.

While the preferred embodiments of the invention have been shown and described, it will be obvious to those skilled in the art that changes, variations and modifications may be made therein without departing from the invention in its broader aspects and, therefore, the aim in the appended claims is to cover such changes and modifications as fall within the scope and spirit of the invention.

WHAT IS CLAIMED IS:

1. A computer implemented software module adaptable to a host operating system to optimize the functional utility of the underlying hardware in the system comprising:

means for collecting performance data of the hardware;

means for accessing critical task selection points within the host operating system; and

said software module being an Autopilot™ software implemented in the hardware and embedded in said means for collecting and said means for allocating to enable prediction of over-commitment of inherent resources of the hardware and manipulate the workload to avoid bottlenecks.

2. The software module of claim 1 wherein said means for collecting performance data includes a PAFS.

3. The software module of claim 1 wherein said means for accessing includes base operating system hooks.

4. A system level tuning and optimization device including a software module which plugs into a host operating system to thereby form a layer in a server toolset architecture, the server architecture implemented software comprising:

hardware components for collecting performance data;

means for providing access to said performance data; and

interface control means to provide access to the tool set architecture;

said software module being an Autopilot™ software to optimally operate and tune the operating system and further having operable data communications with the hardware components, said means for providing access and said interface control means.

5. The device of claim 4 wherein said hardware component include PAFS, EPM, BPM and drivers.

6. A software module implemented in a hardware and forming a layer in a server toolset architecture. the hardware-implemented module comprising:

components of said hardware structured to collect performance data;

a host operating system;

base operating system hooks; and

a computer program running on said operating system;

said based operating system hooks enabling access to the software module such that critical task selection points within said operating system are monitored by the software module to thereby ensure an optimal use of the hardware by said computer program.

7. The software module of claim 1 wherein said module includes an AutoPilot software that plugs directly in said host operating system.

8. The software module of claim 1 wherein said hardware components include PAFS, EPM, BPM and drivers.

9. The software module of claim 3 wherein said EPM includes a set of specialized registers to collect statistics on tasks running on a CPU and their execution behavior thereof.

10. The software module of claim 4 wherein said statistics collected by said registers includes cache hit statistics, instruction pipeline utilization and functional unit utilization.

11. The software module of claim 2 wherein said AutoPilot software is controlled by an AutoPilot control and said control includes tunable variables which are configurable through a standard UnixWare idtune interface.

12. The software module of claim 7 wherein said AutoPilot control includes a tool for system specific tuning.

13. An AutoPilot software implemented in a PA architecture to enhance performance of an underlying hardware, comprising:

a PAFS forming a subsystem of the PA;

a server; and

an application

said application running on said server wherein execution patterns of said application are monitored by the AutoPilot software using real-time performance data collected through said PAFS such that the hardware resources are optimized.

14. The AutoPilot software of claim 9 wherein said PAFS includes a communication with the AutoPilot in the PA architecture.

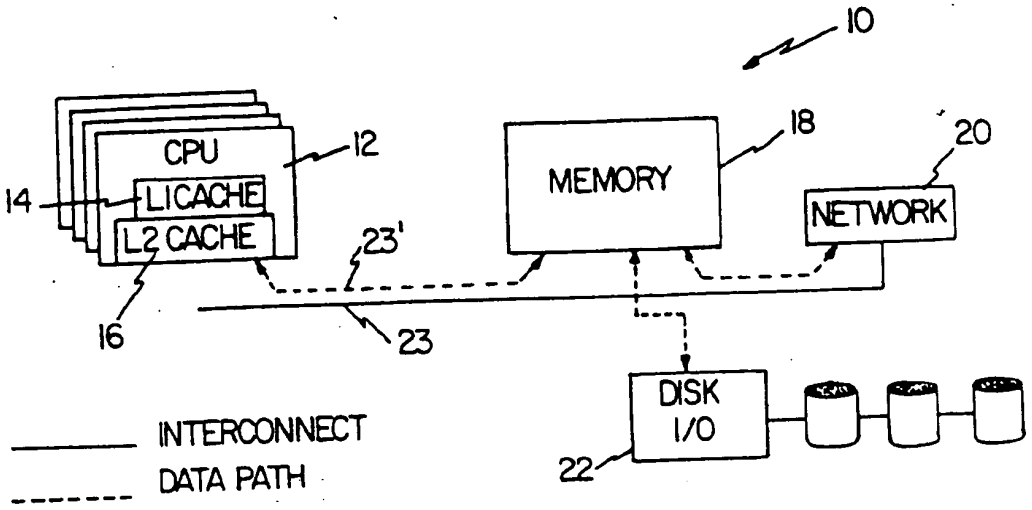

10
CPU
12
14
L1CACHE
L2 CACHE
16
23'
23
MEMORY
18
20
NETWORK
DISK
I/O
22
INTERCONNECT
DATA PATH

Fig. 1

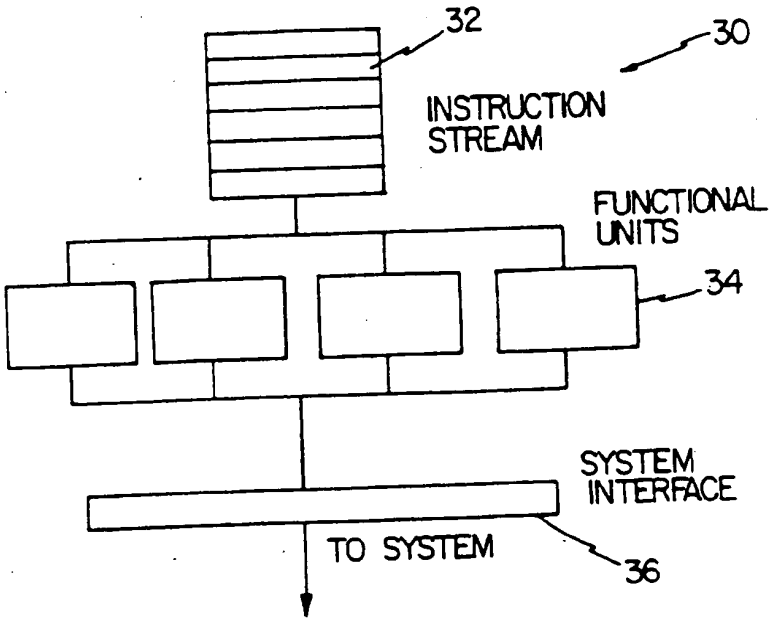

32
30
INSTRUCTION
STREAM
FUNCTIONAL
UNITS
34
SYSTEM
INTERFACE
TO SYSTEM
36

Fig. 2

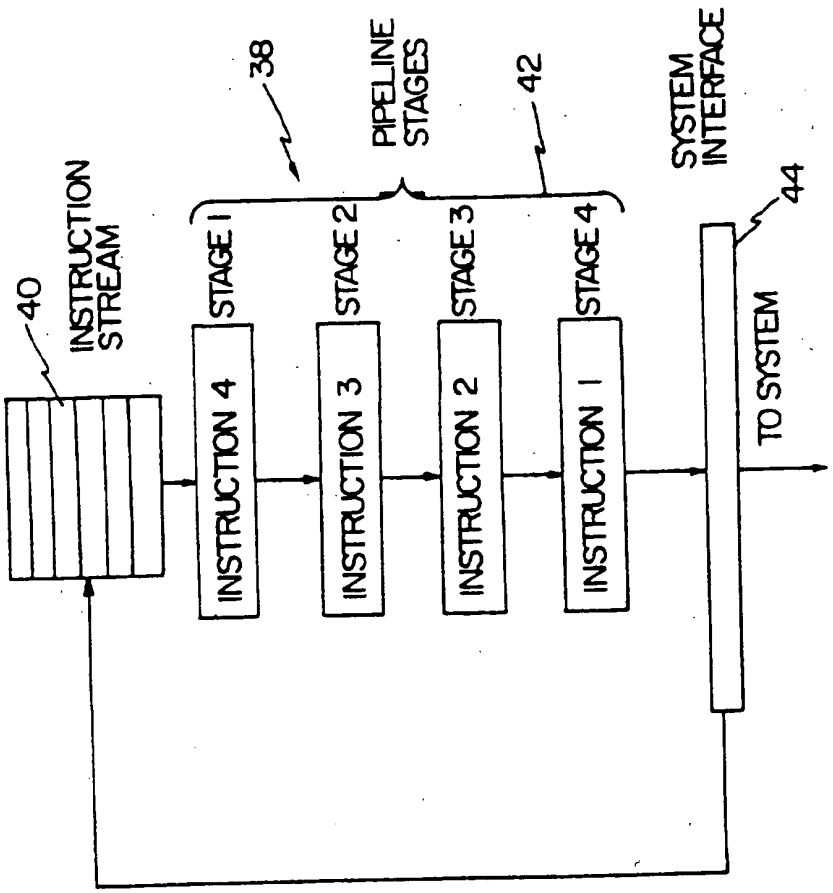

40
INSTRUCTION STREAM
38
PIPELINE STAGES
42
SYSTEM INTERFACE
44
STAGE 1
STAGE 2
STAGE 3
STAGE 4
INSTRUCTION 4
INSTRUCTION 3
INSTRUCTION 2
INSTRUCTION 1
TO SYSTEM

Fig. 3

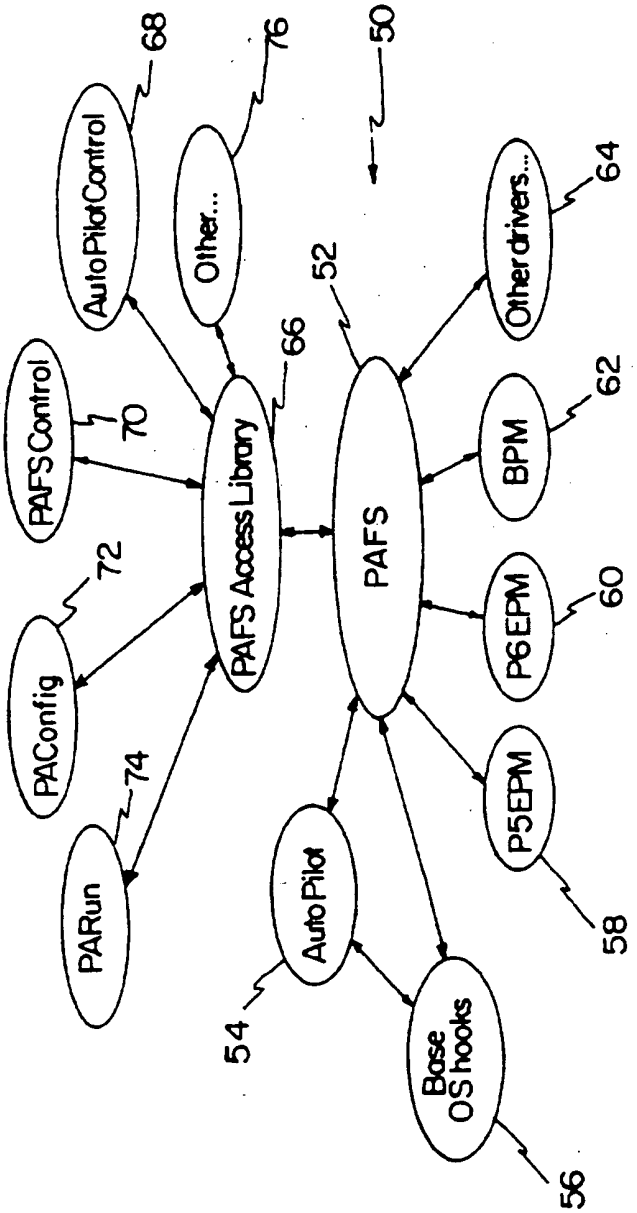

50
52
PAFS
PAFS Access Library
66
AutoPilotControl
68
Other...
76
PAFSControl
70
PAConfig
72
PARun
74
AutoPilot
54
Base OS hooks
56
Other drivers...
64
BPM
62
P6EPM
60
P5EPM
58

Fig 4

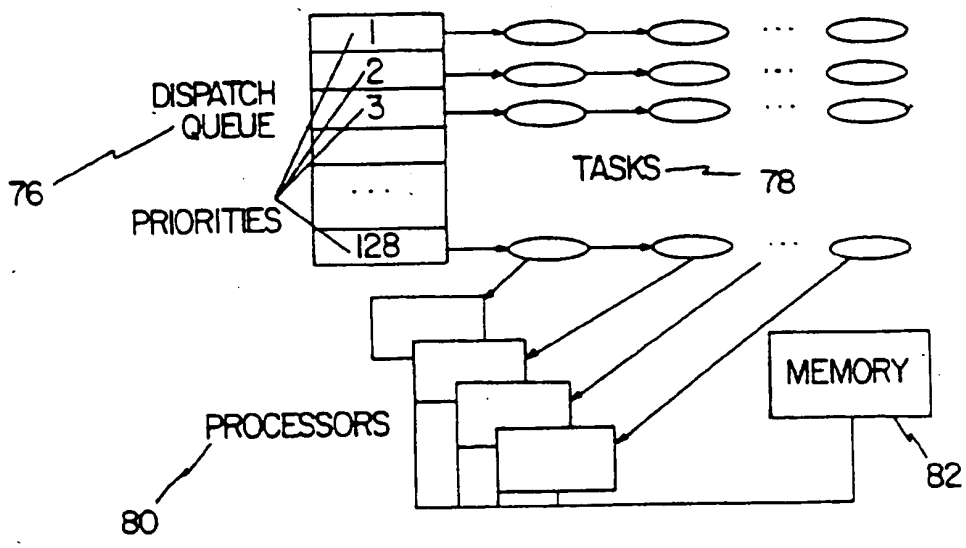

DISPATCH QUEUE
76
PRIORITIES
1
2
3
128
TASKS
78
MEMORY
82
PROCESSORS
80

Fig. 5

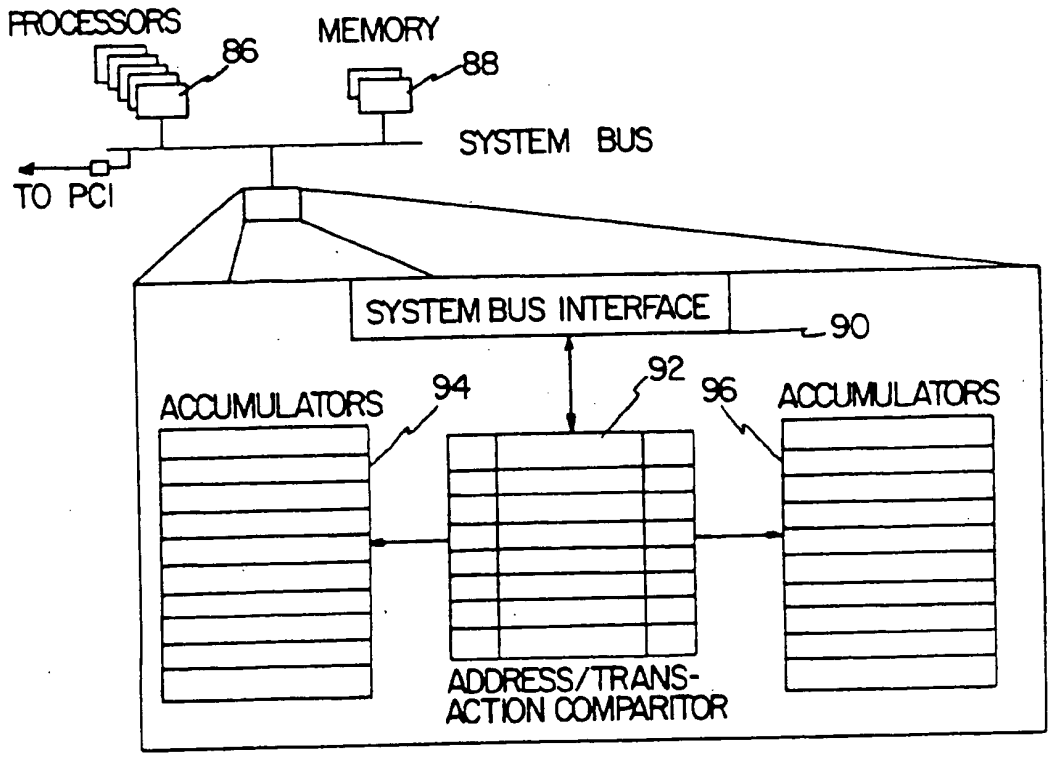

PROCESSORS
86
MEMORY
88
SYSTEM BUS
TO PCI
SYSTEM BUS INTERFACE
90
92
94
96
ACCUMULATORS
ACCUMULATORS
ADDRESS/TRANS-
ACTION COMPARITOR

Fig. 6

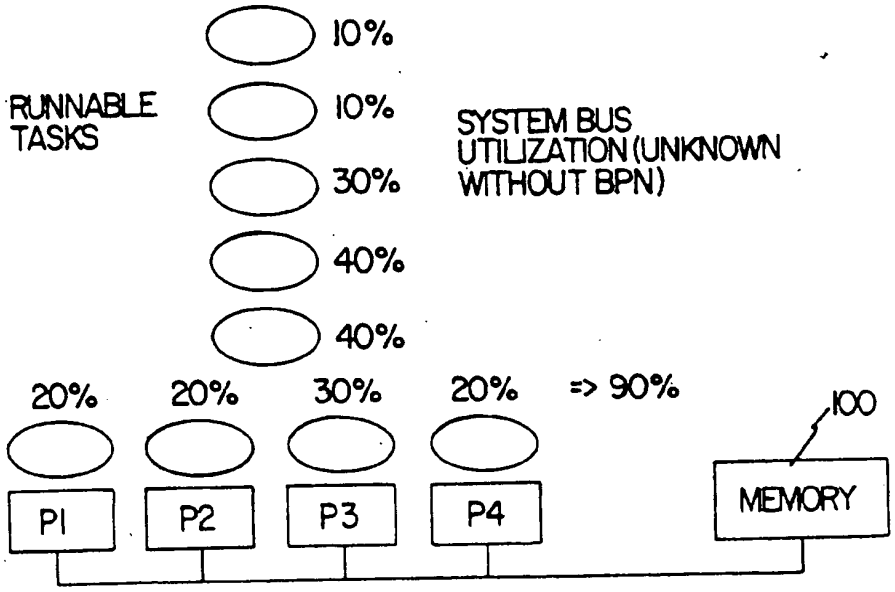

10%
10%
30%
40%
40%
RUNNABLE TASKS
SYSTEM BUS UTILIZATION (UNKNOWN WITHOUT BPN)
20%
20%
30%
20%
=> 90%
100
P1
P2
P3
P4
MEMORY

Fig. 7

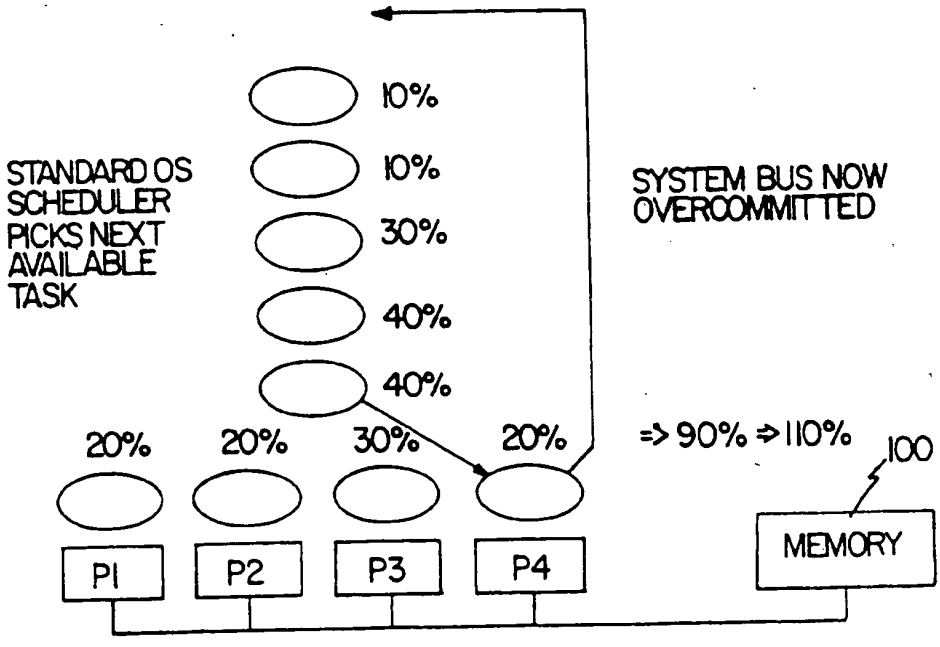

10%
10%
30%
40%
40%
STANDARD OS SCHEDULER PICKS NEXT AVAILABLE TASK
SYSTEM BUS NOW OVERCOMMITTED
20%
20%
30%
20%
=> 90% => 110%
100
P1
P2
P3
P4
MEMORY

Fig. 8

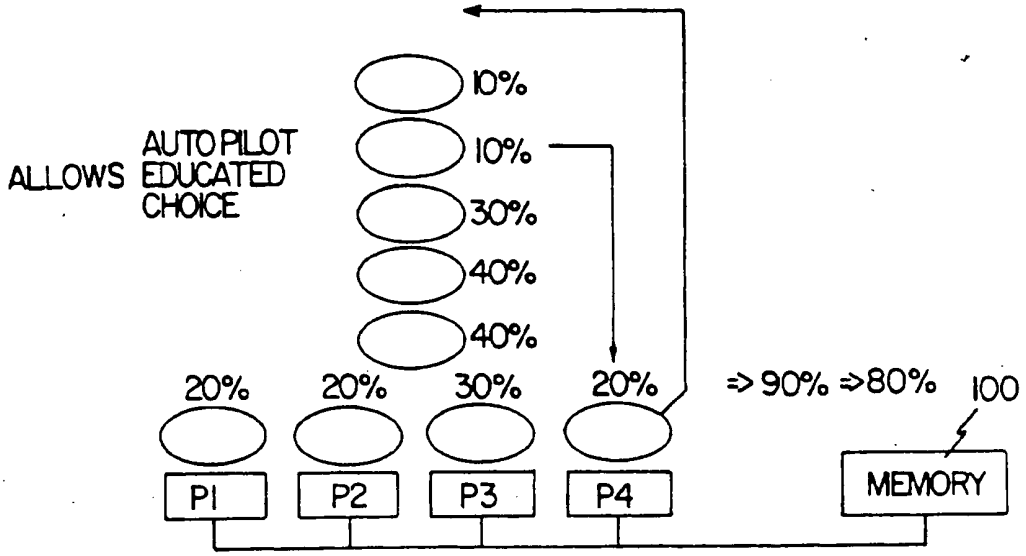

ALLOWS AUTO PILOT EDUCATED CHOICE
10%
10%
30%
40%
40%
20%
20%
30%
20%
=> 90% => 80%
100
MEMORY
P1
P2
P3
P4

Fig. 9A

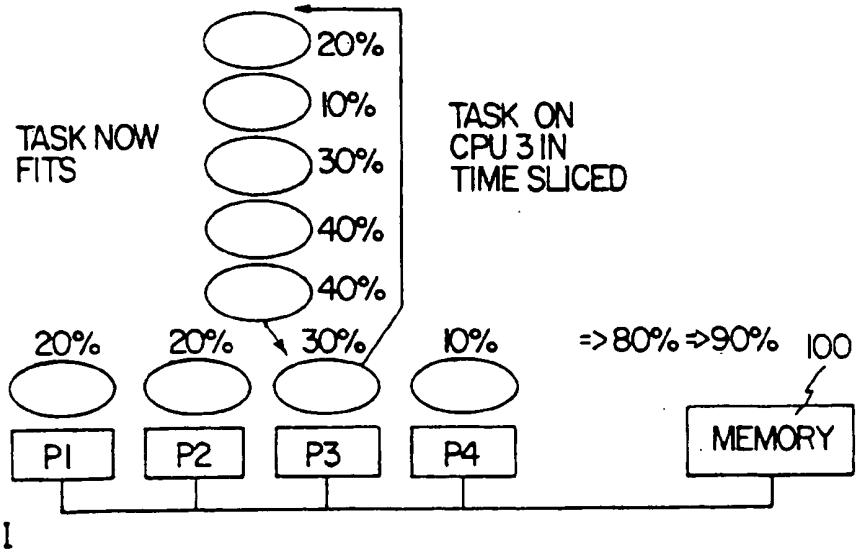

TASK NOW FITS
TASK ON CPU 3 IN TIME SLICED
20%
10%
30%
40%
40%
20%
20%
30%
10%
=> 80% => 90%
100
MEMORY
P1
P2
P3
P4

Fig. 9B

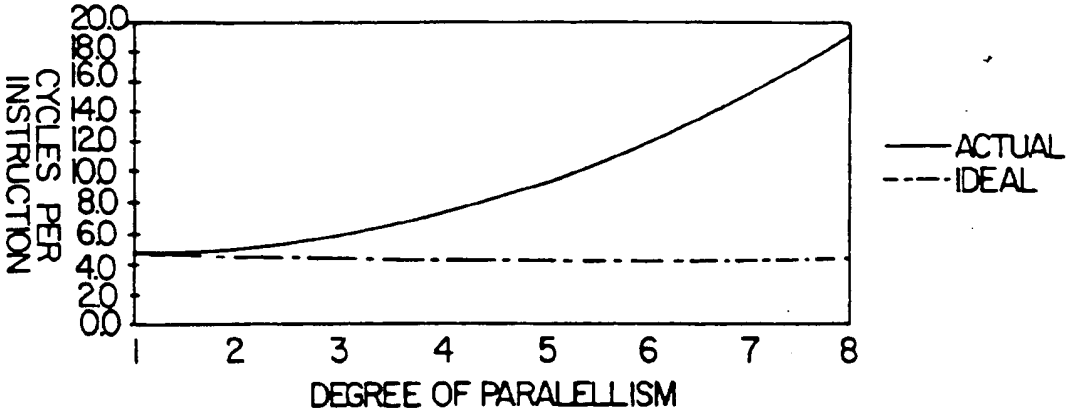


Fig. 10A

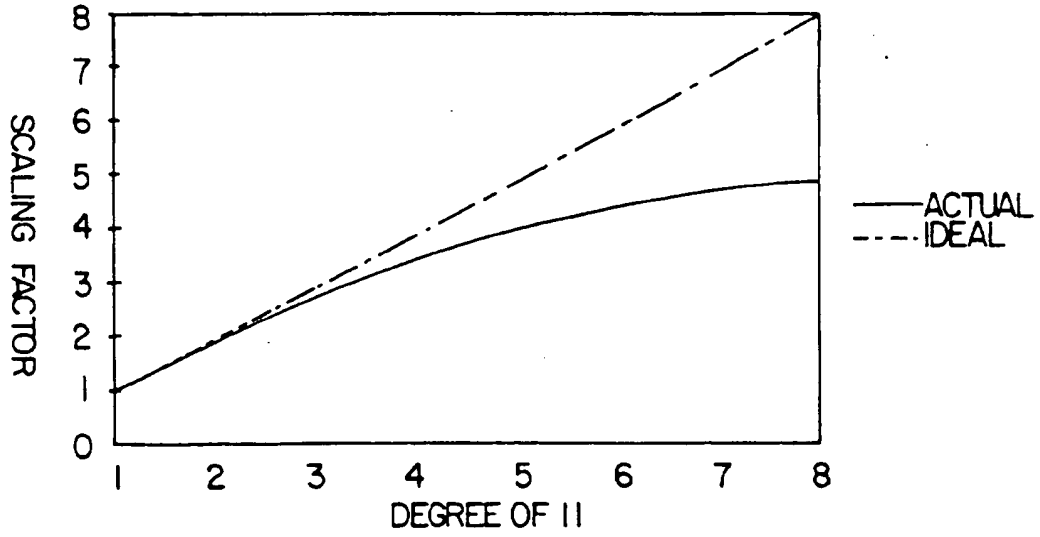


Fig. 10B

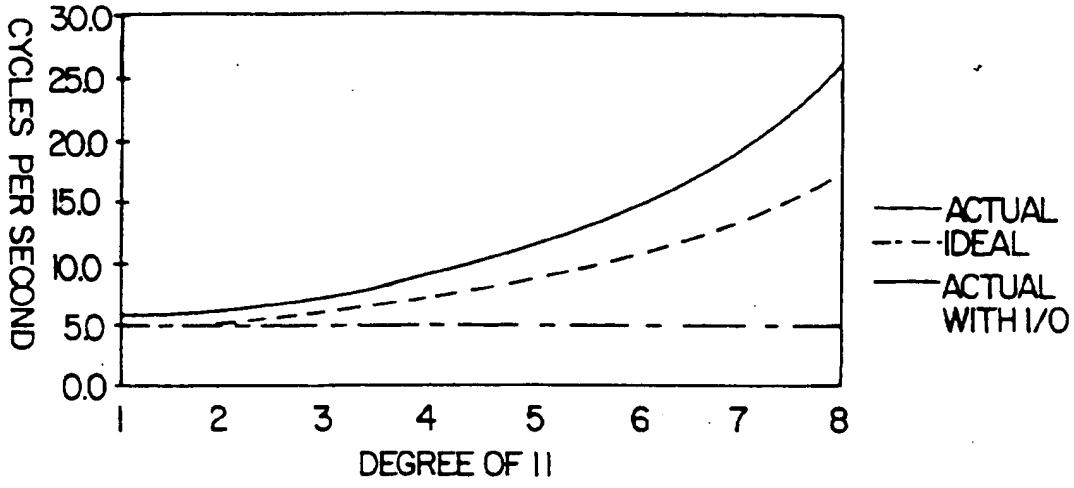


Fig. 10C

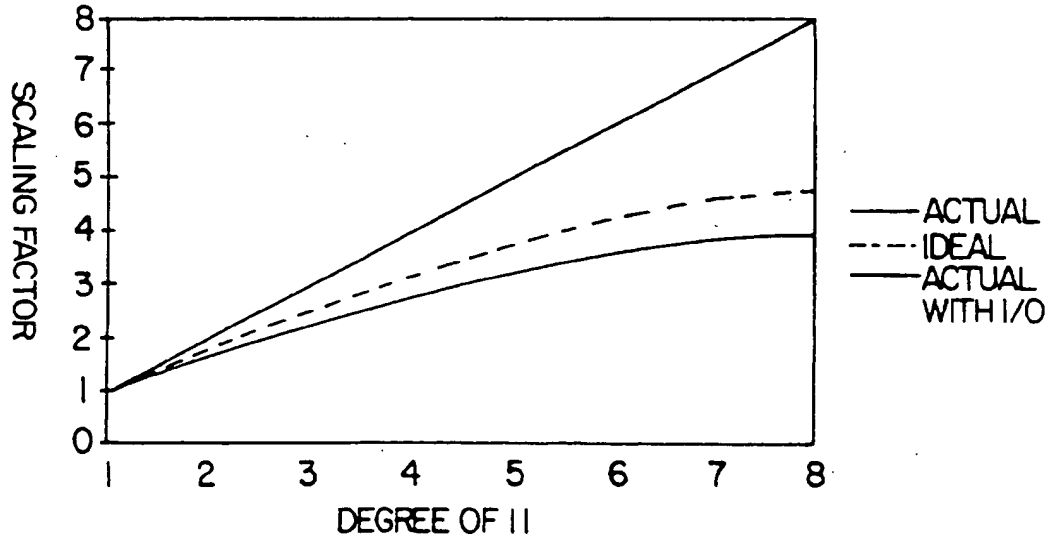


Fig. 10D

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/14540

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) :Please See Extra Sheet.
US CL :395/650
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 395/650;395/700

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

APS, DIALOG

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	Stacks, Volume 3, No. 2, published February 1995, Nemzow, Martin, "Tuning the Unix Environment", pages 1 - 11, especially page 3, lines 26-42, page 5, lines 24-43, page 6, lines 1-19, 3-43, page 7, lines 26-43, page 10 lines 5-43.	1-13
Y	DBMS, Volume 7, No. 6, published 15 June 1994, (no author given), "Tools and Utilities", pages 1-27, especially page 27 lines 30-32.	1, 2, 4, 5, 7, 8, 11, 13, 14

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 04 DECEMBER 1996	Date of mailing of the international search report 31 JAN 1997
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 305-3230	Authorized officer ST. JOHN COURTENAY III Telephone No. (703) 305-9600

Form PCT/ISA/210 (second sheet)(July 1992)*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US96/14540

A. CLASSIFICATION OF SUBJECT MATTER:
IPC (6):

G06F 9/44